GPU 2

CSCI 4850/5850 High-Performance Computing

Spring 2018

Tae-Hyuk (Ted) Ahn

Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University
Learning Objectives

- You will get started how to use GPU CUDA language.
Parallel Programming in CUDA C/C++

• But wait... GPU computing is about massive parallelism!

• We’ll start by adding two integers and build up to vector addition
Moving to Parallel

● GPU computing is about massive parallelism

  ▪ So how do we run code in parallel on the device?

```c
add<<< 1, 1 >>>();

add<<< N, 1 >>>();
```

● Instead of executing `add()` once, execute N times in parallel
Vector Addition on the Device

- With \texttt{add()} running in parallel we can do vector addition.

- Terminology: each parallel invocation of \texttt{add()} is referred to as a block.
  - The set of blocks is referred to as a \textit{grid}.
  - Each invocation can refer to its block index using \texttt{blockIdx.x}.

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

- By using \texttt{blockIdx.x} to index into the array, each block handles a different index.
Vector Addition on the Device

```c
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

<table>
<thead>
<tr>
<th>Block 0</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
</tr>
</thead>
</table>
Vector Addition on the Device: \texttt{add()}

- Returning to our parallelized \texttt{add()} kernel

\begin{verbatim}
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
\end{verbatim}

- Let's take a look at \texttt{main()}...
#define N 10

int main(int argc, char *argv[]) {
    int a[N], b[N], c[N];            // host copies of a, b, c
    int *d_a, *d_b, *d_c;            // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, N*size);
    cudaMalloc((void **)&d_b, N*size);
    cudaMalloc((void **)&d_c, N*size);

    // Setup input values
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
}
Vector Addition on the Device: main()

    // Copy inputs to device
    cudaMemcpy(d_a, &a, N*size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, N*size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, N*size, cudaMemcpyDeviceToHost);

    // Display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free the memory allocated on the GPU
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
#include <iostream>
#include <cstdlib>

#define N 10

using namespace std;

__global__ void add(int *a, int *b, int *c)
{
    int tid = blockIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main(int argc, char *argv[])
{
    int a[N], b[N], c[N];            // host copies of a, b, c
    int *d_a, *d_b, *d_c;            // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)d_a, N*size);
    cudaMalloc((void **)d_b, N*size);
    cudaMalloc((void **)d_c, N*size);

    // Setup input values
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // Copy inputs to device
    cudaMemcpy(d_a, &a, N*size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, N*size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, N*size, cudaMemcpyDeviceToHost);

    // Display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free the memory allocated on the GPU
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory

- Two threads from two different blocks cannot cooperate
Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D, 2D, or 3D \( \text{blockIdx.}\{x,y,z\} \)
  - Thread ID: 1D, 2D, or 3D \( \text{threadIdx.}\{x,y,z\} \)

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...
Built-in Variables for Block and Thread

Built-in variables:
- threadIdx.{x,y,z} – thread ID within a block
- blockIdx.{x,y,z} – block ID within a grid
- blockDim.{x,y,z} – number of threads within a block
- gridDim.{x,y,z} – number of blocks within a grid

kernel<<<nBlocks,nThreads>>>(args)
  Invokes a parallel kernel function on a grid of nBlocks where each block instantiates nThreads concurrent threads
CUDA Threads

• Terminology: a block can be split into parallel threads

• Let’s change `add()` to use parallel threads instead of parallel blocks

```c
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

• We use `threadIdx.x` instead of `blockIdx.x`

• Need to make one change in `main()`…
main ()

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);
Indexing Arrays with Blocks and Threads

• No longer as simple as using `blockIdx.x` and `threadIdx.x`
  – Consider indexing an array with one element per thread (8 threads/block)

```
threadIdx.x           threadIdx.x           threadIdx.x           threadIdx.x
0 1 2 3 4 5 6 7       0 1 2 3 4 5 6 7       0 1 2 3 4 5 6 7       0 1 2 3 4 5 6 7
                  01234567 01234567 01234567 01234567

blockIdx.x = 0   blockIdx.x = 1   blockIdx.x = 2   blockIdx.x = 3
```

• With M threads/block a unique index for each thread is given by:

```c
int index = threadIdx.x + blockIdx.x * M;
```
Indexing Arrays: Example

Which thread will operate on the red element?

```
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

  ```
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  ```

- Combined version of `add()` to use parallel threads and parallel blocks

  ```
  __global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
  }
  ```

- What changes need to be made in `main()`?
main ()

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>> (d_a, d_b, d_c);
#include <iostream>
#include <cstdlib>

#define N (4*8)
#define THREADS_PER_BLOCK 8

using namespace std;

__global__ void add(int *a, int *b, int *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main(int argc, char *argv[])
{
    int a[N], b[N], c[N]; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, N*size);
    cudaMalloc((void **)&d_b, N*size);
    cudaMalloc((void **)&d_c, N*size);

    // Setup input values
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // Copy inputs to device
    cudaMemcpy(d_a, &a, N*size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, N*size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
    //add<<<4,8>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, N*size, cudaMemcpyDeviceToHost);

    // Display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free the memory allocated on the GPU
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

- Sharing Data Between Threads
  - Terminology: within a block, threads share data via `shared memory`
  - Extremely fast on-chip memory, user-managed
  - Declare using `__shared__`, allocated per block
  - Data is not visible to threads in other blocks
__syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block (block level synchronization barrier)
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

```c
__global__ void globFunction(int *arr, int N)
{
    __shared__ int local_array[THREADS_PER_BLOCK]; //local block memory cache
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    //...calculate results
    local_array[threadIdx.x] = results;

    //synchronize the local threads writing to the local memory cache
    __syncthreads();

    // read the results of another thread in the current thread
    int val = local_array[(threadIdx.x + 1) % THREADS_PER_BLOCK];

    //write back the value to global memory
    arr[idx] = val;
}
```
CUDA Thread Indexing Cheatsheet

- [Link](http://cs.calvin.edu/curriculum/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf)